Chapter 3. Writing SOAP Web Services

In Chapter 2, we looked under the hood of SOAP at the XML underneath. In this chapter, we demonstrate how to create, deploy, and use SOAP web services using toolkits for Java, Perl, and Microsoft's new .NET platform. We cover the installation, configuration, and use of SOAP::Lite for Perl, Apache SOAP for Java, and Microsoft .NET for C#.

The task of *creating* and *deploying* web services is really not all that difficult, nor is it all that different than what developers currently do in more traditional web applications. The tendency on all platforms is to automate more and more of the gory details and tedious work in creating web services. Most programmers don't need to know the exact details of encodings and envelopes; instead, they'll simply use a SOAP toolkit such as those described here.

3.1 Web Services Anatomy 101

In Chapter 1, we touched briefly on the fact that a web service consists of three components: a listener to receive the message, a proxy to take that message and translate it into an action to be carried out (such as invoking a method on a Java object), and the application code to implement that action.

The listener and proxy components should be completely transparent to the application code, if properly implemented. The ideal situation in most cases is that the code doesn't even know it is being invoked through a web service interface, but that is not always possible, or desirable.

A good example of a seamless, simple web services implementation is the SOAP::Lite for Perl written by Pavel Kulchenko. This package allows any installed Perl module to be automatically deployed as a web service without any work on the part of the module developer. The proxy can automatically load and invoke any subroutine in any module.

3.1.1 SOAP Implementations and Toolkits

There is a surprisingly long list of SOAP implementations available to developers. In this book, we have chosen to focus on three of the most popular tools: Apache SOAP for Java, SOAP::Lite for Perl, and Microsoft .NET. No matter which toolkit you use, the fundamental process of creating, deploying, and using SOAP web services is the same.

A comprehensive and up-to-date listing of all known SOAP implementations and toolkits can be found by visiting either http://www.soaplite.com/ or http://www.soapware.org/. There are SOAP toolkits for all the popular programming languages and environments (Java, C#, C++, C, Perl, PHP, and Python, just to name a few).

3.1.2 Handling SOAP Messages

The integration of SOAP toolkits varies with the transport layer. Some implement their own HTTP servers. Some expect to be installed as part of a particular web server, so that rather than serving up a web page, the HTTP daemon hands the SOAP message to the toolkit's

proxy component, which does the work of invoking the code behind the web service (see Figure 3-1).

Figure 3-1. The HTTP daemon passes the request to the SOAP proxy, which then invokes the code behind the web service

Still other SOAP toolkits support a pluggable transport mechanism that allows you to select different transport protocols by doing hardly anything more than setting a property value. SOAP::Lite is a good example of this with its support for FTP, HTTP, IO, Jabber, SMTP, POP3, TCP, and MQSeries transports.

Whether the transport is built-in or pluggable, all SOAP toolkits provide the proxy component, which parses and interprets the SOAP message to invoke application code. The proxy must understand how to deal with things like encoding styles, translation of native types of data in to XML (and vice versa), whether headers in the SOAP message that have the `mustUnderstand="true"` flag set are actually understood—basically, everything that is covered in Chapter 2.

When the proxy component is handed a SOAP message by a listener, it must do three things:

1. Deserialize the message, if necessary, from XML into some native format suitable for passing off to the code.
2. Invoke the code.
3. Serialize the response to the message (if one exists) back into XML and hand it back to the transport listener for delivery back to the requester.

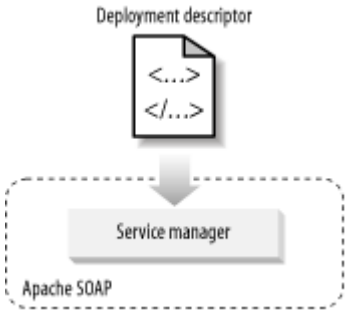
Despite differences in how various SOAP implementations accomplish these tasks, all SOAP web service tools follow this same simple pattern.

3.1.3 Deploying Web Services

Deploying a web service involves telling the proxy component which code to invoke when a particular type of message is received. In other words, the proxy component has to know that a `getQuote` message is going to be handled by the `samples.QuoteServer` Java class or the `QuoteServer.pm` Perl module. Once this has happened, clients can access the server, send the message, and trigger a call to application code.

Web service tools have different deployment mechanisms. SOAP::Lite requires that the Perl module be in `@INC`, Perl's module search path. Apache's SOAP implementation requires a deployment descriptor file, which describes the Java class and rules for mapping Java objects used in the service to their XML equivalents. This file must be added to a deployed services registry used by Apache SOAP (see Figure 3-2).

Figure 3-2. Unlike SOAP::Lite, where the server program contains a description of which modules are to be deployed as services, Apache SOAP uses a separate deployment descriptor file



3.2 Creating Web Services in Perl with SOAP::Lite

Perl, like most languages, hides the programmer from the complexities of SOAP with a toolkit. The SOAP::Lite toolkit is one of the most complete implementations of SOAP available, supporting both Versions 1.1 and 1.2 of SOAP. It has strong support for alternate transports (FTP, HTTP, IO, Jabber, SMTP, POP3, TCP, and MQSeries), which we'll use later to demonstrate SOAP over Jabber.

3.2.1 Installing SOAP::Lite

SOAP::Lite, like many Perl modules, is available on the Comprehensive Perl Archive Network (CPAN). CPAN is a network of web and FTP sites with identical content—the source to thousands of Perl modules. You can access CPAN through a Perl command-line client or via the Web at http://www.cpan.org/. See http://www.cpan.org/misc/cpan-faq.html#How_install_Perl_modules for information on installing Perl modules.

Example 3-1 shows a sample installation of SOAP::Lite using the interactive CPAN command-line shell.

Example 3-1. Installing SOAP::Lite with the CPAN shell

```
C:\book>perl -MCPAN -e shell
cpan shell—CPAN exploration and modules installation (v1.59_54)
cpan> install SOAP::Lite
```

(You may be walked through configuring the CPAN shell if this is the first time you have run it.) The CPAN shell will connect to a CPAN site and download the source for SOAP::Lite. Once downloaded, the shell will attempt to build the module. SOAP::Lite has a series of interactive steps to configure the module, shown in Example 3-2. You can either use a default configuration or manually select from a menu of options to build a custom configuration.

Example 3-2. SOAP::Lite's interactive configuration

```
We are about to install SOAP::Lite and for your convenience will provide
you with list
of modules and prerequisites, so you'll be able to choose only modules you
need for
your configuration.

XMLRPC::Lite, UDDI::Lite, and XML::Parser::Lite are included by default.
Installed
transports can be used for both SOAP::Lite and XMLRPC::Lite.

Client (SOAP::Transport::HTTP::Client)                                [yes]
Client HTTPS/SSL support
   (SOAP::Transport::HTTP::Client, require OpenSSL)                    [no]
Client SMTP/sendmail support (SOAP::Transport::MAILTO::Client)        [yes]
Client FTP support (SOAP::Transport::FTP::Client)                     [yes]
Standalone HTTP server (SOAP::Transport::HTTP::Daemon)                [yes]
Apache/mod_perl server (SOAP::Transport::HTTP::Apache, require Apache)[no]
FastCGI server (SOAP::Transport::HTTP::FCGI, require FastCGI)          [no]
POP3 server (SOAP::Transport::POP3::Server)                           [yes]
IO server (SOAP::Transport::IO::Server)                               [yes]
MQ transport support (SOAP::Transport::MQ)                             [no]
JABBER transport support (SOAP::Transport::JABBER)                     [no]
MIME messages [required for POP3, optional for HTTP]
    (SOAP::MIMEParser)                                                [no]
SSL support for TCP transport (SOAP::Transport::TCP)                  [no]
Compression support for HTTP transport (SOAP::Transport::HTTP)        [no]

Do you want to proceed with this configuration? [yes]
```

In most cases, the default configuration is adequate. We, however, are going to make a slight change to the configuration in order to demonstrate the use of Jabber as a transport protocol for SOAP. To indicate this, answer "no" to the question "Do you want to proceed with this configuration?" Press the Enter key to accept the default options for each of the configuration items until you get to the one that asks whether you plan to use the Jabber transport support module. Answer "yes" and press Enter. The CPAN shell will then make sure that all of the prerequisites and support modules for using Jabber are installed. You may select the default options for the remainder of the installation process.

3.2.2 The Hello Server

No book introducing a new programming system can get by without including a Hello World sample illustrating how easy the system is to use.

Start by creating the Hello World Perl module shown in Example 3-3.

Example 3-3. Hello.pm

```
# Hello.pm - simple Hello module
package Hello;
sub sayHello {
  shift;                              # remove class name
  return "Hello " . shift;
}
1;
```

This module will be the code that sits behind our web service interface. There are several approaches you can take with SOAP::Lite to deploy this module as a web service.

If you already have a CGI-capable web server (and most people do) you can simply create the CGI script shown in Example 3-4.

Example 3-4. hello.cgi

```
#!/usr/bin/perl -w
# hello.cgi - Hello SOAP handler
use SOAP::Transport::HTTP;
SOAP::Transport::HTTP::CGI
 -> dispatch_to('Hello::(?:sayHello)')
 -> handle
;
```

This CGI script is the glue that ties the listener (the HTTP server daemon) to the proxy (the SOAP::Lite module). With this glue, SOAP::Lite will dispatch any received request to the Hello World module's `sayHello` operation.

Perl will need to find the Hello module, though. If you don't have permission to install Hello into one of Perl's default module directories (print `@INC` to see what they are), use the `lib` pragma to tell Perl to look in the directory containing the Hello module. If the module is in `/home/pavel/lib` then simply add this `use` line to `hello.cgi`:

```
use lib '/home/pavel/lib';
```

Your SOAP web service is deployed and ready for action.

3.2.3 The Hello Client

To test your Hello web service, simply use the client script in Example 3-5.

Example 3-5. hw_client.pl

```
#!/usr/bin/perl -w
# hw_client.pl - Hello client
use SOAP::Lite;
my $name = shift;
print "\n\nCalling the SOAP Server to say hello\n\n";
print "The SOAP Server says: ";
print SOAP::Lite
  -> uri('urn:Example1')
  -> proxy('http://localhost/cgi-bin/helloworld.cgi')
  -> sayHello($name)
  -> result . "\n\n";
```

Running this script should give you the following results:

```
% perl hw_client.pl James

Calling the SOAP Server to say hello
The SOAP Server says: Hello James
%
```

We see here a complete SOAP web service. Granted, it doesn't do much, but that wasn't the point. The point was that the process we followed (create the code, deploy the service, invoke the service) is the same regardless of the service we're implementing, the tools we're using, or the complexity of the service.

3.2.4 A Visual Basic Client

To prove that it really is SOAP that we're passing around here, the Visual Basic Script in Example 3-6 uses the Microsoft XML Parser's ability to send XML directly over HTTP to exchange SOAP messages back and forth with the Hello World service.

Example 3-6. hw_client.vbs

```
Dim x, h
Set x = CreateObject("MSXML2.DOMDocument")
x.loadXML  "<s:Envelope  xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'
xmlns
:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/
XMLSchema'><s:Body><m:sayHello                    xmlns:m='urn:Example1'><name
xsi:type='xsd:string'>James</
name></m:sayHello></s:Body></s:Envelope>"
msgbox x.xml, , "Input SOAP Message"
Set h = CreateObject("Microsoft.XMLHTTP")
h.open "POST", "http://localhost:8080"
h.send (x)
while h.readyState <> 4
wend
msgbox h.responseText,,"Output SOAP Message"
```

Running the Visual Basic script should demonstrate two things to you: invoking SOAP web services is easy to do, and it doesn't matter which language you use. Perl and Visual Basic strings are being interchanged over HTTP.

In the next example, there are two messages exchanged between the requester and the service provider. The request, encoding the service we're calling (`sayHello`) and the parameter (`James`), is shown in Example 3-7, and the response containing `Hello James` is shown in Example 3-8.

Example 3-7. Hello request

```
<s:Envelope
  xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <s:Body>
    <m:sayHello xmlns:m='urn:Example1'>
      <name xsi:type='xsd:string'>James</name>
    </m:sayHello>
  </s:Body>
</s:Envelope>
```

Example 3-8. Hello response

```
<s:Envelope
  xmlns:s="http://www.w3.org/2001/06/soap-envelope"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <s:Body>
    <n:sayHelloResponse xmlns:n="urn:Example1">
      <return xsi:type="xsd:string">
         Hello James
      </return>
    </n:sayHelloResponse>
  </s:Body>
</s:Envelope>
```

3.2.5 Changing Transports

SOAP::Lite supports many transport protocols. Let's modify the Hello World sample so that it can be invoked using Jabber. This demonstrates the modular nature of the web services stack, where the packaging can be independent of the transport. You might deploy a web service over Jabber to take advantage of the presence and identity features that Jabber provides.

Create an instance of the SOAP-aware Jabber server built into SOAP::Lite using the script in Example 3-9.

Example 3-9. sjs, the SOAP Jabber server

```
#!/usr/bin/perl -w
# sjs - soap jabber server

use SOAP::Transport::JABBER;

my $server = SOAP::Transport::JABBER::Server
  -> new('jabber://soaplite_server:soapliteserver@jabber.org:5222')
  -> dispatch_to('Hello')
;

print "SOAP Jabber Server Started\n";
do { $server->handle } while sleep 1;
```

Then, modify the client script we used earlier to point to the Jabber address of the service, as shown in Example 3-10.

Example 3-10. sjc, the SOAP Jabber client

```
#!/usr/bin/perl -w
# sjc - soap jabber client

use SOAP::Lite;

my $name = shift;
print "\n\nCalling the SOAP Server to say hello\n\n";
print "The SOAP Server says: ";
print SOAP::Lite
  -> uri('urn:Example1')
  -> proxy('jabber://soaplite_client:soapliteclient@jabber.org:5222/' .
           'soaplite_server@jabber.org/')
```

```
  -> sayHello($name)
  -> result . "\n\n";
```

The `soaplite_server` and `soaplite_client` accounts are registered with *Jabber.org*, so this example should work as typed. To avoid confusion when everyone reading this book tries the example at the same time, you should register your own Jabber IDs at http://www.jabber.org/.

Now, in case you're curious as to how Jabber is capable of carrying SOAP messages, Example 3-11 is the text of the `sayHello` message sent by the previous script. As you can see, the SOAP message itself is embedded into the Jabber message element. This demonstrates the flexibility of both protocols.

Example 3-11. Jabber message with SOAP payload

```
<iq to="soapproxy@johndoe.ibm.com/soaprouter" id="6" type="get">
  <query xmlns="soap-message">
    <s:Envelope
       xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <s:Body>
        <m:sayHello xmlns:m="urn:Example1">
           <name xsi:type="xsd:string">James</name>
        </m:sayHello>
      </s:Body>
    </s:Envelope>
  </query>
</iq>
```

3.3 Creating Web Services in Java with Apache SOAP

Creating web services in Java is more work than in Perl with SOAP::Lite, but the process is essentially the same. To illustrate how it's done, let's create the same Hello World web service and deploy it using the Apache SOAP tools.

Apache SOAP is the Apache Software Foundation's implementation of the SOAP protocol. It is designed to run as a servlet within any Java HTTP Server. As such, it implements only the proxy part of the message handling process. Like SOAP::Lite, Apache SOAP's list of features is impressive, sharing many of the same benefits as its Perl-based counterpart.

3.3.1 Installing Apache SOAP

Apache SOAP can be used as both a client and provider of SOAP web services. A server-side installation of Apache SOAP involves placing some *.jar* files in your classpath. You will need a separate web server that supports Servlets and Java Server Pages, such as Apache's Tomcat (http://jakarta.apache.org/tomcat/).

The Apache SOAP homepage, http://xml.apache.org/soap/index.html, has links to both source-only and precompiled distributions of the toolkit. Installing the precompiled binary distribution is as simple as downloading a Zip archive and extracting it into a directory.

On the client, three *.jar* files from the distribution (*soap.jar*, *mail.jar*, and *activation.jar*) must be present in your classpath. Also present must be any Java API for XML Parsing (JAXP) aware XML parser, such as Xerces Version 1.4 (http://xml.apache.org/xerces-j/).

Assuming that you installed Apache SOAP *.jar* files in the *C:\book\soap* directory, set your SOAP_LIB environment variable to *C:\book\soap\lib*. Adding the *.jar* files to your classpath then entails:

```
set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\soap.jar
set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\mail.jar
set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\activation.jar
```

Or, in the Unix Bourne shell (*/bin/sh*):

```
CLASSPATH = $CLASSPATH;$SOAP_LIB/soap.jar
CLASSPATH = $CLASSPATH;$SOAP_LIB/mail.jar
CLASSPATH = $CLASSPATH;$SOAP_LIB/activation.jar
```

The exact steps for a server installation will depend on which web application server you are using, but the process is essentially the same. The first step is to ensure the same three *.jar* files are located in your application server's classpath.

If your application server supports the use of web application archives (WAR files), simply use the *soap.war* file that ships with Apache SOAP. Apache Tomcat supports this. The Apache SOAP documentation includes detailed installation instructions for Tomcat and a number of other environments.

If you intend to use the Bean Scripting Framework (BSF) to make script-based web services, you need to ensure that *bsf.jar* and *js.jar* (a BSF JavaScript implementation) are also in the web application server's classpath.

The vast majority of problems encountered by new Apache SOAP users are related to incorrect classpaths. If you encounter problems writing web services with Apache SOAP, be sure to start your debugging by checking your classpath!

3.3.2 The Hello Server

We're going to do the same things we did in Perl: create the code, deploy the service, and use the service. Example 3-12 shows the Java code for the Hello class.

Example 3-12. Hello.java

```
package samples;
public class Hello {
   public String sayHello(String name) {
       return "Hello " + name;
   }
}
```

Compile the Java class and put it somewhere in your web server's classpath.

3.3.3 Deployment Descriptor

Next we must create a *deployment descriptor* to tell the Apache SOAP implementation everything it needs to know to dispatch `sayHello` messages to the `samples.Hello` class. This is shown in Example 3-13.

Example 3-13. Deployment descriptor for samples.Hello

```
<dd:service             xmlns:dd="http://xml.apache.org/xml-soap/deployment"
id="urn:Example1">
 <dd:provider type="java"
              scope="Application"
              methods="sayHello">
    <dd:java class="samples.Hello"
             static="false" />
 </dd:provider>
 <dd:faultListener>
  org.apache.soap.server.DOMFaultListener
 </dd:faultListener>
 <dd:mappings />
</dd:service>
```

The information contained within a deployment descriptor is fairly basic. There is the class name of the Java code being invoked (`<dd:java class="samples.Hello" static="false" />`), and an indication of the session scope of the service class (application or session scope, as defined by the Java Servlet specification), an indication of which `faultListener` to use (used to declare how faults are handled by the SOAP engine), and a listing of Java-to-XML type mappings. We will demonstrate later how the type mappings are defined.

Apache SOAP supports the use of *pluggable providers* that allow web services to be implemented not only as Java classes, but as Enterprise Java Beans, COM Classes, and Bean Scripting Framework scripts. Full information about how to use pluggable providers is available in the documentation and not covered here.

While simple in structure, deployment descriptor files must be created for every web service that you want to deploy. Thankfully, there are tools available that automate that process, but they still require the developer to walk through some type of wizard to select the Java class, the methods, and the type mappings. (A *type mapping* is an explicit link between a type of XML data and a Java class, and the Java classes that are used to serialize or deserialize between those types.)

Once the file is created, you have to deploy it with the Apache SOAP service manager. There are two ways to do this: you can use the Service Manager Client or, if you're using the XML-based Service Manager that ships with Apache SOAP, modify the deployment registry directly.

The first method requires executing the following command:

```
%              java              org.apache.soap.server.ServiceManagerClient
http://hostname:port/soap/servlet/
rpcrouter deploy foo.xml
```

Where *`hostname:port`* is the hostname and port that your web service is listening on.

One interesting fact you should notice here is that the Apache Service Manager is itself a web service, and that deployment of a new service takes place by sending a SOAP message to the server that includes the deployment descriptor. While this is handy, it's not necessarily all that secure (considering the fact that it would allow anybody to deploy and undeploy services on your web server). To disable this, set the `SOAPInterfaceEnabled` option in the *soap.xml* configuration file to `false`. This will prevent the `ServiceManagerClient` from working.

The second approach will only work if you're using the XML Configuration Manager. This component allows you to store deployment information in an XML file. This file is located in the *web-apps* folder where your Apache SOAP servlet is located.

The XML is nothing more than a root element that contains all of the deployment descriptors for all of the services deployed. To deploy the Hello World service, simply take the deployment descriptor we wrote earlier and append it to this list. The next time that the SOAP servlet is started, the service manager will be reinitialized and the new service will be ready for use. A sample configuration file is given in Example 3-14.

Example 3-14. Apache SOAP configuration file

```
<root>
  <dd:service xmlns:dd="http://xml.apache.org/xml-soap/deployment"
              id="urn:Example1">
    <dd:provider type="java"
                 scope="Application"
                 methods="sayHello">
       <dd:java class="samples.Hello"
                static="false" />
    </dd:provider>
    <dd:faultListener>
     org.apache.soap.server.DOMFaultListener
    </dd:faultListener>
    <dd:mappings />
   </dd:service>
</root>
```

3.3.4 The Hello Client

To invoke the Hello World service, use the Java class in Example 3-15.

Example 3-15. Hello client in Java

```
import java.io.*;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class Example1_client {

  public static void main (String[] args)
       throws Exception {

    System.out.println("\n\nCalling the SOAP Server to say hello\n\n");
    URL url = new URL (args[0]);
    String name = args[1];
```

```
    Call call = new Call (  );
    call.setTargetObjectURI("urn:Example1");
    call.setMethodName("sayHello");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC;);
    Vector params = new Vector (  );
    params.addElement (new Parameter("name", String.class, name, null));
    call.setParams (params);

    System.out.print("The SOAP Server says: ");

    Response resp = call.invoke(url, "");

    if (resp.generatedFault (  )) {
      Fault fault = resp.getFault (  );
      System.out.println ("\nOuch, the call failed: ");
      System.out.println ("  Fault Code   = " + fault.getFaultCode (  ));
      System.out.println ("  Fault String = " + fault.getFaultString (  ));
    } else {
      Parameter result = resp.getReturnValue (  );
      System.out.print(result.getValue (  ));
      System.out.println(  );
    }
  }
}
```

The amount of code to accomplish this relatively simple operation may seem surprising (nine lines to actually initialize and invoke the web services call). Java will never be as terse as Perl and other scripting languages, but it has other strengths. Also, various Java-based SOAP toolkits such as The Mind Electric's GLUE and IBM's Web Services ToolKit support dynamic proxy interfaces that cut down the amount of code necessary to invoke web services. Those interfaces, however, generally require additional mechanisms, such as WSDL, to simplify the programming interface. We will take a look at these dynamic proxies later in Chapter 5. For now, if you compile and run this class, you'll end up with the same result that we saw in the Perl example:

```
% java samples.Hello http://localhost/soap/servlet/rpcrouter James

Calling the SOAP Server to say hello
The SOAP Server says: Hello James

%
```

Your Java web service is finished. If you have both the Perl and Java versions installed, run the Perl client script again but point it at the Java version of the Hello World service (the modified script is shown in Example 3-16). You'll see that everything still works.

Example 3-16. hw_jclient.pl, the Perl client for the Java Hello World server

```
#!/usr/bin/perl -w
# hw_jclient.pl - java Hello client
use SOAP::Lite;
my $name = shift;
print "\n\nCalling the SOAP Server to say hello\n\n";
print "The SOAP Server says: ";
print SOAP::Lite
  -> uri('urn:Example1')
  -> proxy('http://localhost/soap/servlet/rpcrouter James')
  -> sayHello($name)
  -> result . "\n\n";
```

Which will produce the expected result:

```
% perl hw_client.pl James

Calling the SOAP Server to say hello
The SOAP Server says: Hello James

%
```

3.3.5 The TCPTunnelGui Tool

One very useful tool that comes bundled with Apache SOAP is TCPTunnelGui, a debugging tool that lets a developer view the SOAP messages that are being sent to and returned from a SOAP web service. The tool is a proxy—it listens on the local machine and forwards traffic to and from the real SOAP server. The contents of the messages passing through the local port will be displayed in the graphical interface.

Launch the tool by typing:

```
% java org.apache.soap.util.net.TcpTunnelGui listenport tunnelhost tunnelport
```

`Listenport` is the local TCP/IP port number you want the tool to open and listen to requests. *`Tunnelhost`* is the address of the server (either DNS or IP address) where the traffic is to be redirected, and *`tunnelport`* is the port number at tunnelhost.

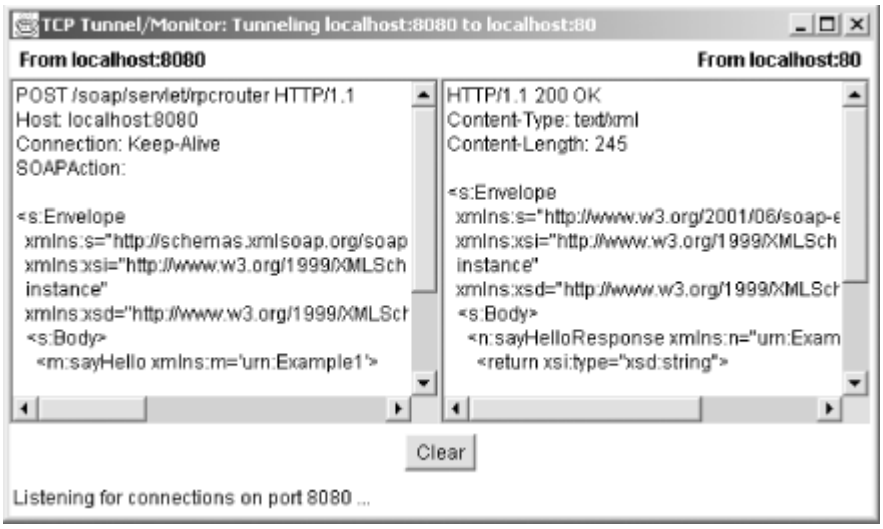
For example, assume your Hello World service is deployed at *http://www.example.com/soap/servlet/rpcrouter*. To view the messages sent to and from that service by redirecting traffic through local TCP/IP port 8080, launch the TCPTunnelGui tool with the following parameters:

```
% java org.apache.soap.util.net.TcpTunnelGui 8080 http://www.example.com 80
```

And now direct the Hello World SOAP requests to `http://localhost:8080/soap/servlet/rpcrouter`.

Figure 3-3 shows TCPTunnelGui displaying the SOAP messages for each request to the Hello World service.

Figure 3-3. The TCPTunnelGui Tool showing the SOAP messages sent to and from the Hello World service



TCPTunnelGui is an extremely valuable tool for anybody wanting to learn how SOAP Web services work (or debugging why a service doesn't work!).

3.4 Creating Web Services In .NET

For web service developers working strictly on the Windows platform, Microsoft's .NET development platform offers built-in support for easily creating and deploying SOAP web services. Let's walk through how you create the Hello World service using C#, the new Java-like programming language designed specifically for use with .NET.

3.4.1 Installing .NET

The first thing you need to do is download and install the Microsoft .NET SDK Beta 2 from http://msdn.microsoft.com/. This free distribution contains everything you need to create and run any .NET application, including .NET Web Services.

There are, however, several prerequisites that you need:

1. You must be running Windows 2000, Windows NT 4.0, Windows 98, or Windows Millennium Edition.
2. You must have Microsoft Internet Explorer Version 5.01 or higher.
3. You must have the Microsoft Data Access Components (Version 2.6 or higher) installed.
4. And you must have Microsoft Internet Information Server (IIS) installed and running. .NET Web services can only be deployed within the IIS environment.

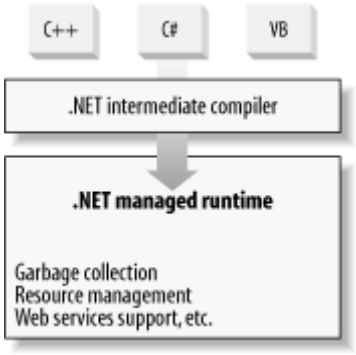
The .NET Framework SDK installation is a fairly automatic process, with an easy-to-use installation wizard. Once installed, we can create the Hello World service.

3.4.2 Introducing .NET

Before we get into exactly how web services are created in .NET, let's take a quick walk through the .NET architecture to help put things into perspective.

First and foremost, .NET is a runtime environment similar to the Java Virtual Machine. Code packages, called *assemblies*, can be written in several .NET specific versions of popular programming languages like Visual Basic, C++, C#, Perl, Python, and so on. Assemblies run within a managed, hierarchically organized runtime called the "Common Language Runtime" that deals with all of the low-level memory and system management details (see Figure 3-4).

Figure 3-4. The .NET managed runtime



Currently, .NET runs basically as an extension to the existing COM environment upon which the current versions of Windows are built. As such, .NET can be utilized anywhere COM can be used, including within Microsoft's Internet Information Server (IIS) environment.

.NET web services are specific types of .NET assemblies that are specially flagged for export as web services. These assemblies are either contained within or referenced from a new type of server-side script called an *.asmx* file. The .NET extensions to IIS recognize files ending in *.asmx* as web services and automatically export the functions of the referenced assemblies.

The process is simple:

1. Write the code.
2. Save the code in an *.asmx* file.
3. Move the *.asmx* file to your IIS web server.
4. Invoke your web service.

3.4.3 Saying Hello

.NET introduces a programming language called C#. We'll develop our example web service in C#, but remember that .NET makes it just as easy to develop in Visual Basic, C++, and other languages.

Example 3-17 defines the .NET Hello World service. You can use an ordinary text editor to create this file.

Example 3-17. HelloWorld.asmx, a C# Hello World Service

```
<%@ WebService Language="C#" Class="Example1" %>

using System.Web.Services;

[WebService(Namespace="urn:Example1")]
public class Example1 {

    [ WebMethod ]
    public string sayHello(string name) {
        return "Hello " + name;
    }

}
```

Notice how similar the code looks to the Java version we created earlier. At heart, a function that appends two strings isn't rocket science. The bracketed sections (`<% %>` and `[ ]`) tell the .NET runtime that this code is intended to be exported as a SOAP web service.

The `<% WebService Language="C#" Class="Example1" %>` line tells .NET that we are exporting one web service, written in C#, implemented by the `Example1` class.

The `using` line imports a module, in this case the standard web services classes.

The `[WebService(Namespace="urn:Example1")]` line is optional, but allows us to declare various attributes of the web service being deployed. In this instance, we are setting an explicit namespace for the web service rather than allowing .NET to assign a default (which, by the way, will always be `http://tempuri.org/`). Other attributes you can set for the web service include the name and textual description of the service.

The `[ WebMethod ]` line sets an attribute that flags the methods in the class to be exposed as part of the web service. As with the `WebService` attribute previously, we could use this line to define various custom properties about the web service operation. Options include whether to buffer the response of the operation; if buffered, how long to keep it buffered; whether to maintain a session state for the operation; whether transactions are supported on the method; what the exported name of the operation is; and a textual description of the operation. In the case of the Hello World example, we have no need to set any of these options, so we simply leave it alone.

What will .NET do with all of this information? It's actually quite simple. Whenever a *.asmx* file is requested by a client through IIS, the .NET runtime will first compile the code for the service if it hasn't done so already. The compiled code is temporarily cached in memory and recompiled every time a change is made to the *.asmx* file or the IIS server is restarted.

Next, the .NET runtime will determine what type of request is being made. There are several choices:

1. The request may be for information about the web service.
2. The request may be for information about one of the methods exported by the web service.

3. Or, the request may be to invoke an operation on the web service. .NET allows the operations to be invoked one of three different ways: through an HTTP-GET operation, through an HTTP-POST operation, or through the use of SOAP messages. .NET is one of the only web services platforms that allow web services to be invoked using multiple protocols.

3.4.4 Deploying the Service

Save the *HelloWorld.asmx* file to a location in your IIS web root. Take note of the *.asmx* file's URL. For example, if your Microsoft IIS server is installed at *c:\inetpub* (the default installation location), the web root is *c:\inetpub\wwwroot*. If you saved the *.asmx* file directly to this location, the URL of the *.asmx* file will be *http://localhost/helloworld.asmx*, where *localhost* is the DNS name or IP address of your IIS server. Once you've completed this step, your .NET web service is deployed.

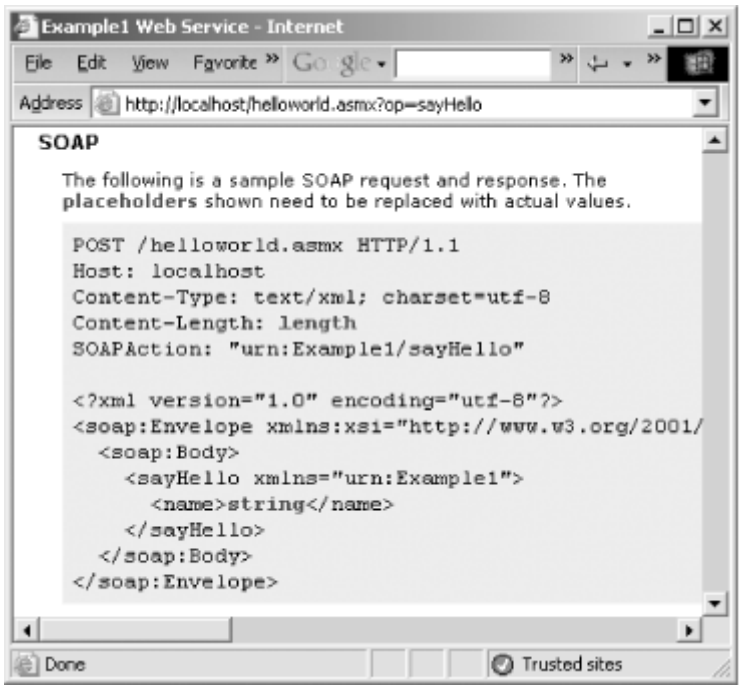
Ensure that your .NET environment and web service are fully operational by launching your favorite web browser and navigating to *http://localhost/HelloWorld.asmx*. If all goes well, you should be presented with an automatically generated HTML page that documents the Hello World service you just created (see Figure 3-5).

Figure 3-5. Automatically generated documentation for the .NET web service

These pages are generated dynamically whenever an HTTP-GET request is received for the deployed *.asmx* file. You do not have to do anything to create these pages.

Clicking on the "sayHello" link will yield a detailed description of how to invoke the `sayHello` operation using SOAP, HTTP-GET, and HTTP-POST, as well as a simple HTML form for testing the operation (see Figure 3-6).

Figure 3-6. Auto-generated documentation for the sayHello operation



To test the service, either type your name in the test form at the top of the automatically generated documentation page (see Figure 3-7), or navigate your browser to *http://localhost/helloworld.asmx/sayHello?name=yourname*.

Figure 3-7. Ensure that the service works using the Test form

Either method should generate the response shown in Figure 3-8.

Figure 3-8. A typical HTTP-GET web service response

If you get the "Hello James" message, you're ready to move on.

3.4.5 Invoking the Service Using SOAP

Creating a SOAP client for the Hello World service using .NET is, surprisingly, harder than creating the service itself. There are tools to make it easier (we will explore them briefly in Chapter 5), but for now we'll go through the steps manually so you know what is going on.

Again using your favorite text editor, create *HelloWorld.cs* (the *.cs* extension indicates C# source code) from Example 3-18.

Example 3-18. HelloWorld.cs, a C# HelloWorld client

```
// HelloWorld.cs

using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.Web.Services;

[System.Web.Services.WebServiceBindingAttribute(
     Name="Example1Soap",
     Namespace="urn:Example1")]
public class Example1 :
             System.Web.Services.Protocols.SoapHttpClientProtocol {

    public Example1(  ) {
        this.Url = "http://localhost/helloworld.asmx ";
    }

    [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
         "urn:Example1/sayHello",
         RequestNamespace="urn:Example1",
         ResponseNamespace="urn:Example1",
         Use=System.Web.Services.Description.SoapBindingUse.Literal,

ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public string sayHello(string name) {
        object[] results = this.Invoke("sayHello",
                                       new object[] {name});
        return ((string)(results[0]));
    }
```

```
    public static void Main(string[] args) {
       Console.WriteLine("Calling the SOAP Server to say hello");
       Example1 example1 = new Example1(  );
       Console.WriteLine("The SOAP Server says: " +
                         example1.sayHello(args[0]));
    }
}
```

The `[System.Web.Services.WebserviceBindingAttribute]` line tells the .NET managed runtime that this particular .NET assembly is going to be used to invoke a web service. When the assembly is compiled, .NET will automatically supply the infrastructure to make the SOAP request work.

Subclassing `System.Web.Services.Protocols.SOAPHttpClientProtocol` tells the .NET runtime which protocol you want to use (SOAP over HTTP in this case). Within the constructor for this class, set the URL for the web service (the assignment to *this.Url*).

The rest of the class declares a proxy for the `sayHello` operation, specifies various attributes of the web services invocation, calls the invoke method, and returns the result.

Lastly, we create the main entry point for the C# application. The entry point does nothing more than create an instance of our client class and invoke the proxy `sayHello` operation, outputting the results to the console.

Compile the client to a *HelloWorld.exe* application:

```
C:\book>csc HelloWorld.cs
```

To invoke the web service, simply type:

```
C:\book>HelloWorld yourname
```

You will be greeted with the same result we saw previously with the Java and Perl versions of the Hello World service:

```
Calling the SOAP Server to say hello
The SOAP Server says: Hello James
```

3.5 Interoperability Issues

At the time of this writing, .NET's SOAP implementation still has a few issues that need to be worked out, primarily in the area of interoperability.

Slight variations between the way .NET implements SOAP and SOAP::Lite's implementation of SOAP, for example, cause some difficulty in allowing the two to work together out of the box. To illustrate the problem, follow the steps shown here. One would think that everything would work fine, but it doesn't. I'll point out why after we walk through it.

First, launch the Java TcpTunnelGui tool that ships with Apache SOAP, specifying port 8080 as the local listening port, and redirecting to whatever server you have your *HelloWorld.asmx* file deployed to:

```
C:\book>start java org.apache.soap.util.net.TcpTunnelGui 8080
        localhost 80
```

Then, modify the Perl Hello World client to point to the *HelloWorld.asmx* file, but replace the server part of the URL with `localhost:8080`.

When you run the Perl script:

```
C:\book>perl hello_client1.pl James
```

The result is not what you would expect. The script ends without ever displaying the "Hello James" result. If you take a look at the TcpTunnelGui tool, you'll see that the SOAP message is sent, but the .NET runtime rejects the request and issues a SOAP fault in response. This is shown in Example 3-19.

Example 3-19. SOAP fault from .NET

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>
          System.Web.Services.Protocols.SoapException: Server did
          not recognize the value of HTTP Header SOAPAction:
          urn:Example#sayHello.
          at    System.Web.Services.Protocols.SoapServerProtocol.Initialize(
)
          at  System.Web.Services.Protocols.ServerProtocolFactory.Create(
          Type type, HttpContext context, HttpRequest request,
          HttpResponse response)
      </faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

.NET requires that the HTTP `SOAPAction` header be used to exactly identify the operation on which service is being invoked. .NET requires the format of the `SOAPAction` header to be the service namespace, followed by a forward slash, followed by the name of the operation, or `urn:Example/sayHello`. Notice, though, that SOAP::Lite's default is to use a pound sign (`#`) to separate the service namespace from the name of the operation. This wasn't an issue when we were invoking Java services with SOAP::Lite because Apache SOAP simply ignores the `SOAPAction` header altogether.

To fix this problem, we must explicitly tell SOAP::Lite how to format the SOAPAction header. To do so, make the change to the client script highlighted in Example 3-20.

Example 3-20. Fragment showing change to Perl client script

```
print SOAP::Lite
  -> uri('urn:Example1')
  -> on_action(sub{sprintf '%s/%s', @_ })
  -> proxy('http://localhost:8080/helloworld/example1.asmx')
  -> sayHello($name)
```

```
  -> result . "\n\n";
```

The `on_action` method in SOAP::Lite allows the developer to override the default behavior and specify a new format for the `SOAPAction` header.

However, even with this change there's still a problem. The script will appear to run, but rather than returning the expected `Hello James` string, all that will be returned is `Hello`. The name is missing from the response! This happens because .NET requires all parameters for a method call to be named and typed explicitly, whereas Perl does not do this by default.

Again, take a look at the TcpTunnelGui tool and look at the SOAP message sent to the `HelloWorld.asmx` service from SOAP::Lite. This is shown in Example 3-21.

Example 3-21. The Perl-generated SOAP request sent to the .NET service

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>
    <namesp1:sayHello xmlns:namesp1="urn:Hello">
       <c-gensym3 xsi:type="xsd:string">
          James
       </c-gensym3>
    </namesp1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice the oddly named `c-gensym3` element that contains the input parameter. Because Perl is a scripting language that does not support strong typing or strict function signatures, method parameters do not have names, nor do they have types. When SOAP::Lite creates the SOAP message it automatically generates an element name and sets all parameters to the `string` data type. .NET doesn't like this behavior. If the C# method is written to take a `String` parameter called `name` it expects to find an element in the SOAP envelope called `name` with a type of `xsi:type="xsd:string"`. In XML, that would be as shown in Example 3-22.

Example 3-22. A SOAP request encoded by .NET

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>
    <namesp1:sayHello xmlns:namesp1="urn:Hello">
       <name xsi:type="xsd:string">
          James
       </name>
    </namesp1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The .NET beta also did not properly recognize that the `name` element is declared as part of the same namespace as its parent `sayHello` element. This is a standard rule of XML namespaces. To get SOAP::Lite working with .NET, we must tell SOAP::Lite the name, type, and namespace of each of the parameters we are passing into the operation, as shown in Example 3-23.

Example 3-23. Perl client modified to work with .NET

```
use SOAP::Lite;

my $name = shift;

print "\n\nCalling the SOAP Server to say hello\n\n";
print "The SOAP Server says: ";

print SOAP::Lite
  -> uri('urn:Example1')
  ->on_action(sub{sprintf '%s/%s', @_ })
  ->proxy('http://localhost:8080/helloworld/example1.asmx')
  ->sayHello(SOAP::Data->name(name => $name->type('string')
                        ->uri('urn:Example1'))
  ->result . "\n\n";
```

Now, run the script and you will see that everything works as expected.

Developers who are writing and using web services that may be accessed by a wide variety of SOAP implementations need to be aware that inconsistencies like this will exist between the various toolkits and you need to be prepared to deal with them. As web services become more complex and more mission critical, it is important to have a clear understanding of how to manage these issues. Over time, the more popular SOAP implementations will be honed to a point where they will work together seamlessly, but with many of these implementations still being released as beta and sometimes alpha code status, you must be aware that issues will exist. Luckily, as we will see in Chapter 5, there are workarounds available for some of these problems.